The PowerPC 600 series, part 7: Atomic memory access and cache coherency

devblogs.microsoft.com/oldnewthing/20180814-00

August 14, 2018

Raymond Chen

On the PowerPC 600 series, memory accesses to suitably-aligned locations by a single register are atomic,[1] meaning that even in the face of a conflicting operation on another processor, the result will be the entire previous value or the entire final value, never a mix of the two.

To perform atomic update operations (load-modify-store, also known as interlocked operations), you use the `lwarx` and `stwcx.` instructions:

```
lwarx   rd, ra/0, rb      ; load rd from ra/0 + rb and reserve
stwcx.  rd, ra/0, rb      ; store rd conditionally to ra/0 + rb, update cr0
```

Note that the only supported addressing mode is `x`. No plain instruction, and no `u` forms.

The `lwarx` instruction loads a word and creates a reservation which monitors the memory for changes. Any modification to that address or an address nearby causes the reservation to be lost. The definition of "nearby" is left up to the processor.

The `stwcx.` instruction tries to store *rd* to memory. The store will succeed if the reservation is still in effect and the store is to the same address as the most recent `lwarx`. The result of the operation is reported in the *eq* bit of *cr0*: *eq* is set on success and clear on failure. The instruction also updates the other bits of *cr0* by clearing the *lt* and *gt* bits and capturing the summary overflow bit.

Note that the `stwcx.` instruction ends with a dot because it implicitly updates *cr0*. There is no undotted form.

Regardless of whether the store succeeded, the reservation is cleared.

If you attempt to store back to a location different from the most recent preceding `lwarx`, and the reservation is still valid, the store might or might not succeed, and the *eq* bit will be unpredictable; it need not reflect the actual success of the store. So don't do that.[2]

If you've seen the other RISC architecture atomic operations, this should feel very familiar. Here's a sample interlocked increment:

```
    ; atomically increment the word stored at address r3
loop:
    lwarx   r4, 0, r3         ; load with reservation
    addi    r4, r4, 1         ; increment
    stwcx.  r4, 0, r3         ; store conditional
    bne-    loop              ; if failed (unlikely), try again
    ; on exit r4 contains incremented value
```

You are allowed to abandon a reservation. For example, a compare-exchange starts with a reservation, but if the value is incorrect, it just gives up without ever storing anything.

```
    ; if the word at r3 is equal to r4, then replace it with r5
loop:
    lwarx   r6, 0, r3         ; load with reservation
    cmpw    r6, r4            ; contains correct value?
    bne-    stop              ; if not, then give up
    stwcx.  r5, 0, r3         ; store conditional
    bne-    loop              ; if failed (unlikely), try again
stop:
    ; r6 contains previous value stored at r3
```

As noted above, simple accesses to suitably-aligned locations are atomic, and you can use the `lwarx` / `stwcx.` instructions to construct more complex atomic operations, but none of those instructions impose any memory ordering. In practice, the interlocked operations will usually erect a memory barrier before and/or after the atomic update.

```
    sync                      ; full memory barrier
    isync                     ; acquire
    lwsync                    ; release
```

The `sync` instruction is a full memory barrier.

The `isync` instruction officially discards prefetch, but that has a side effect of preventing future memory operations from starting (because they were discarded), which is effectively an acquire. You usually use it after taking a lock, so that reads intended to be under the lock do not get advanced to before the lock is taken.

The `lwsync` waits for preceding loads and stores to complete, but allows future loads to start. You usually use it just before releasing a lock, so that all accesses that were intended to be protected by the lock are finished before the lock is dropped.

And then there's this guy:

```
    eieio                     ; enforce in-order execution of I/O
```

This instruction is so famous it has its own Wikipedia page. Somebody worked really hard to backronym that mnemonic. It's intended as a memory barrier for memory-mapped I/O, but it is generally useful as well. It acts like a lightweight `lwsync` : It ensures that all pending stores are completed, but it does not prevent future loads from starting or force preceding loads to complete. You can use this just before exiting a lock if the purpose of the lock was to update some data rather than to read some data. The compiler, of course, doesn't usually have this level of insight into your code, so you're unlikely to see this in practice.

There are other types of barriers but you're not likely to encounter them. There are also special instructions to tell the processor that you've written new code to memory, so it should discard any prefetch or instruction cache.

When reading code, you don't need to worry too much about the distinctions between these different types of barriers. You can assume that the compiler used the correct barrier. (Well, unless you're chasing a compiler bug.)

The PowerPC permits implementations to have separate I-cache and D-cache, so you cannot assume that writing code to memory will immediately take effect at execution. You have to explicitly tell the processor that instructions have changed. This is mostly relevant only for jitters, so I won't go into details. I never had to debug a jitter on this guy, and even if I were called upon to do it, I'd just assume that whoever wrote the memory barrier stuff knew what they were doing.

Next time, we'll look at control flow instructions and their absurd mnemonics.

[1] Although not available in little-endian mode, there are instructions in big-endian mode that can load and store multiple registers. Each individual register access is atomic if suitably aligned, but the entire operation is not.

[2] Interrupts and traps do not clear the reservation. This means that if the operating system wants to perform a context switch, it needs to perform a `stwcx.` to a harmless location to force the reservation to be cleared. Otherwise, the thread being switched to might be in the middle of an atomic operation, and its `stwcx.` might succeed based on the previous thread's reservation! This is a rare case where you will intentionally perform a `stwcx.` to an address that doesn't match the preceding `lwarx` .

Raymond Chen

Follow